Programmation Temps Réel

Coordination/Synchronisation

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Novembre 2017

Plan

Plan

- Mécanismes de communication
- Allocation mémoire
  - Notion de pool mémoire
- Mécanismes de synchronisation
  - Drapeau événement
- Notion de section critique
- Gestion des sections critiques
  - Mutex, sémaphore
  - Variables condition
- Problème de l'inversion de priorité
  - Héritage de priorité (PIP)

Introduction à la synchronisation (1/2)

- Les tâches temps-réel doivent souvent se synchroniser et se coordonner lors de l'utilisation de ressources communes.
  - Une tâche doit attendre qu'un traitement se termine.
  - Une tâche doit attendre qu'une ressource matérielle (ou logicielle) se libère.
  - Les tâches doivent s'exécuter dans un ordre précis (notion de précédence).

Introduction à la synchronisation (2/2)

- La synchronisation entre les tâches nécessite un objet particulier qui permet de gérer cette synchronisation.
  - Il s'agit d'un objet utilisé par l'ensemble des tâches concernées par la synchronisation; cet objet est capable de "réveiller" une ou plusieurs tâches en présence d'un événement (généré par une tâche ou une ISR).
  - Chaque objet dispose de sa propre file d'attente.
    - Plusieurs tâches peuvent utiliser le même objet de synchronisation.
  - Chaque objet de synchronisation dispose de sa propre file d'attente (FIFO ou "par priorité").
    - La politique de la file d'attente peut être celle de l'ordonnanceur.
  - Le mode de réveil des tâches peut consister à réveiller une seule tâche (mode normal), ou à réveiller l'ensemble des tâches en attente (mode broadcast).

Objets de synchronisation

- Drapeau événement
- Mutex
- Sémaphore
- Variable condition

Drapeau événement (1/2)

- Le drapeau événement (event flag) permet simplement de signaler aux tâches la présence d'un événement.
  - L'événement peut être quelconque, défini par le développeur.
    - Par exemple, une ISR est activée lors d'une interruption en provenance d'un périphérique, et peut ainsi informer une tâche en attente que le périphérique est prêt pour livrer des données.
- Un drapeau événement est associé à un événement unique
  - Un groupe d'événements (présents ou non) peut être représenté à l'aide d'un tableau de bits (bitmap).
    - Le bit correspondant à un événement particulier est mis à 1 pour "dire" que l'événement s'est produit, ou à 0 dans le cas contraire.
    - Le masque d'événement peut contenir jusqu'à 255 événements (dépendant du noyau).
    - Les tâches peuvent être réveillées en présence d'un événement parmi plusieurs (disjonction), ou seulement si tous les événements sont présents (conjonction).
  - Le mode de réveil est toujours le mode broadcast.
  - La remise à zéro d'un événement se fait de manière explicite.

API Xenomai - Service de drapeau événement

Fonction	Description	Kern.	User
`int rt_event_create(RT_EVENT *event, const char *name, unsigned long ivalue, int mode)`	Création d'un masque d'événements	✓	✓
`int rt_event_signal(RT_EVENT *event, unsigned long mask)`	Mise à un d'un ou plusieurs événements	✓	✓
`int rt_event_wait(RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)`	Attente sur un ou plusieurs des événements	✓	✓
`int rt_event_clear((RT_EVENT *event, unsigned long mask, unsigned long *mask_r)`	Remise à zéro d'un ou plusieurs événements	✓	✓
`int rt_event_delete(RT_EVENT *event)`	Destruction d'un masque d'événements	✓	✓

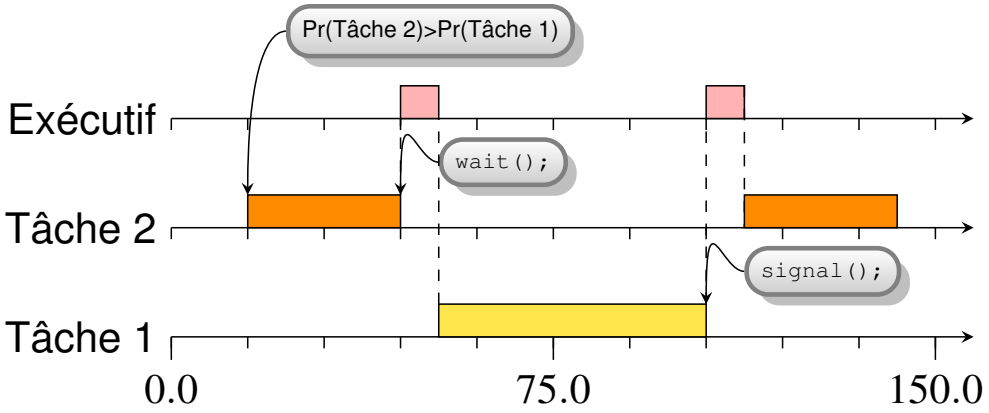
Synchronisation par drapeau événement

Tâche 1

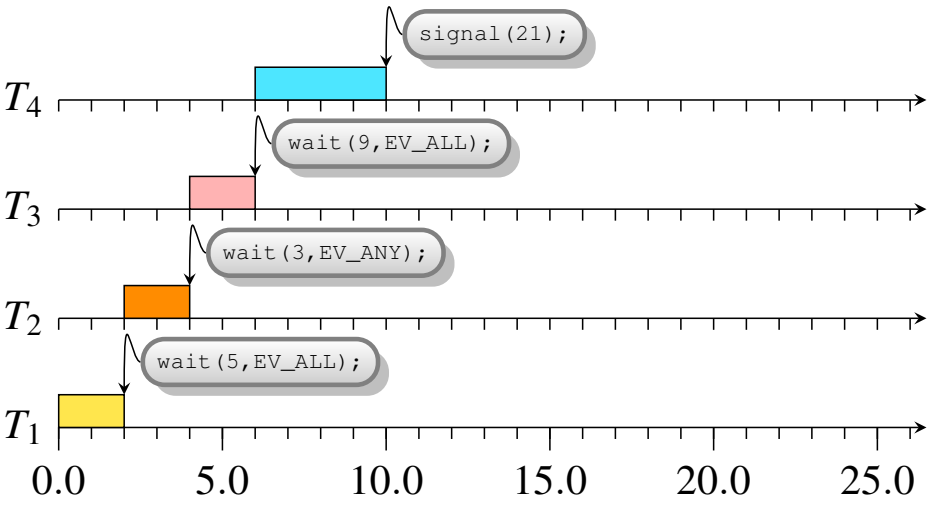
```
void taskT1(void *cookie) {
  int evtMask;
  ...
  evtMask = 9;
  rt_event_signal(events, evtMask);
  ...
}
```

Tâche 2

```
void taskT2(void *cookie) {
  int evtMask;
  ...
  evtMask |= 8;
  rt_event_wait(events, evtMask);
  ...
}
```

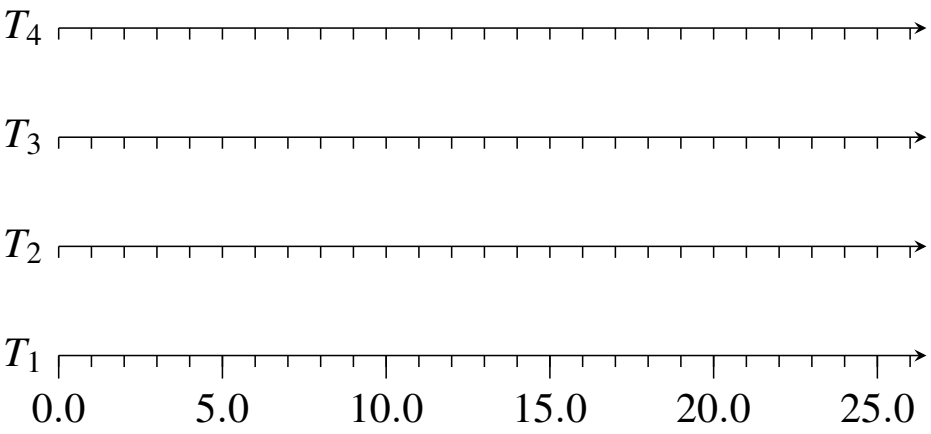


Drapeaux événements - Exemple



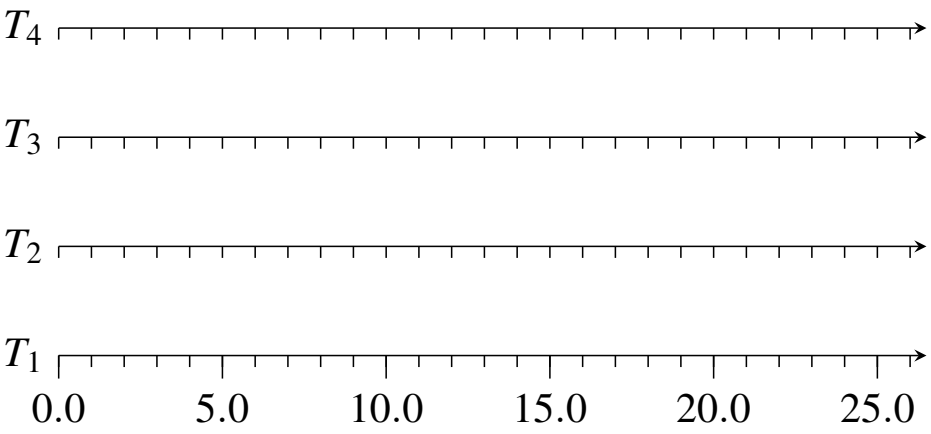
- $C(T_4) = C(T_3) = C(T_2) = C(T_1) = 4$
- $Pr(T_4) < Pr(T_3) < Pr(T_2) < Pr(T_1)$
- Que se passe-t-il à partir du temps 10?

Drapeaux événements - Exemple (1)



- $C(T_4) = C(T_3) = C(T_2) = C(T_1) = 4$
- $Pr(T_4) > Pr(T_3) > Pr(T_2) > Pr(T_1)$
- T_1 effectue `signal(8)` après 2 slots
- T_2 effectue `wait(7,EV_ANY)` après 2 slots
- T_3 effectue `wait(13,EV_ALL)` après 2 slots
- T_4 effectue `signal(5)` après 2 slots

Drapeaux événements - Exemple (2)



- $C(T_4) = C(T_3) = C(T_2) = C(T_1) = 4$
- $Pr(T_4) > Pr(T_3) > Pr(T_2) > Pr(T_1)$
- T_1 effectue `signal(8)` après 2 slots
- T_2 effectue `wait(7,EV_ANY)` après 2 slots et `clear(1)` après 3 slots
- T_3 effectue `wait(13,EV_ALL)` après 2 slots
- T_4 effectue `signal(5)` après 2 slots. Périodique de période 16

Section critique

- Souvent, plusieurs tâches utilisent des ressources communes (matérielles ou logicielles). Il faut donc gérer l'accès simultané (ou concurrent) à ces ressources afin de garder un état cohérent.
  - Ce moyen est appelé exclusion mutuelle.
  - Toutes les instructions manipulant une ou plusieurs ressources partagées forment une section critique.
- La protection des sections critiques se fait au moyen d'objets de synchronisation prévus à cet effet.
  - Mutex
  - Sémaphores
  - Variables condition

Accès concurrent - Exemple

Global

```
static int count=0;
```

```
void taskT1(void *cookie) {
  while (1) {

    /* Accès à une ressource */
    /* commune */
    count = count + 1;

    /* Accès à
un périphérique */
    iowrite(GPIO_1, "blabla");

    rt_task_wait_period(NULL);
  }
}
```

```
void taskT2(void *cookie) {
  int value;
  while (1) {

    /* Accès à une ressource */
    /* commune */
    count = count + 1;

    /* Accès à
un périphérique */
    value=ioread(GPIO_1);

    rt_task_wait_period(NULL);
  }
}
```

Accès concurrent - Explication

- Que signifie `count=count+1;` ?
- En langage machine:

```
mov count,r1    % copie la valeur de la mémoire
                % commune désignée par count
                % dans un registre local r1;

addi 1,r1       % ajoute 1 au registre;

mov r1,count    % stocke le contenu du registre
                % local r1 à l'adresse de
                % count
```


Accès concurrent - Explication

Exemple d'une exécution erronée

Tâche 1	$count = 10$	Tâche 2
$r1 \leftarrow 10$		
	changement de contexte	
		$r1 \leftarrow 10$
		$r1 \leftarrow r1 + 1$
		$count \leftarrow r1$ vaut 11
	changement de contexte	
$r1 \leftarrow r1 + 1$		
$count \leftarrow r1$		
	$count$ vaut ainsi 11 et non 12!	

Section critique - Approche (1/2)

- On pourrait désactiver l'entrée des interruptions à l'entrée d'une section critique.
  - Plus d'interruption du timer système $\Rightarrow$ l'ordonnanceur n'ayant plus la main, il n'y a plus de changement de contexte (changement de tâche).
    - Utilisable seulement sur des sections critiques de petite taille et peu fréquemment appelées.
    - Gros risques de manquer des interruptions importantes et d'introduire des décalages temporels critiques.
- Lors de l'accès à une section critique, la priorité pourrait être augmentée.
  - Seule la tâche de plus haute priorité peut accéder à la ressource.
    - Gros risques de saboter le schéma d'ordonnancement des tâches!

Section critique - Approche (2/2)

- Avant d'entrer dans une section critique, on pourrait appeler une fonction spéciale, qui nous bloque si une autre tâche a déjà accédée cette section critique.

```
/* Au départ, verrou est initialisé à 0 */
void request(int *verrou){
    while (*verrou) ; /* Attente */
    *verrou = 1;
}
```

```
void release(int *verrou) {
    *verrou = 0; /* Libération */
}
```

- Cette solution impose que la fonction request() soit indivisible; cela nécessite des instructions spéciales.
- Cette solution est applicable seulement si l'ordonnançeur peut préempter la tâche en cours d'exécution après un certain temps d'exécution sous peine d'un blocage final.

Algorithme de Peterson

- Plusieurs algorithmes permettent de faire de l'attente active
  - Exemple: Algorithme de Peterson (ici pour 2 tâches)

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

void *Tache0(void *arg)
{
  while (true) {
    intention[0] = true;
    tour = 1;
    while (intention[1] && tour == 1)
      ;
    /* section critique */
    intention[0] = false;
    /* section non-critique */
  }
}
```

```
void *Tache1(void *arg)
{
  while (true) {
    intention[1] = true;
    tour = 0;
    while (intention[0] && tour == 0)
      ;
    /* section critique */
    intention[1] = false;
    /* section non-critique */
  }
}
```

- Applicable seulement si l'ordonnançeur peut préempter la tâche en cours
- Plusieurs problèmes apparaissent: on consomme beaucoup de temps CPU inutilement et il existe un risque de famine pour les tâches moins prioritaires.

Mutex

- Le mutex est un mécanisme simple de verrou permettant la protection d'une section critique.
- En anglais: *mutex*, pour mutual exclusion
  - Lorsqu'une tâche désire accéder la section critique, elle doit d'abord acquérir le verrou. Celui-ci doit être ouvert, c-à-d qu'aucune autre tâche se trouve dans la section critique.
    - La tâche devient (momentanément) propriétaire du mutex.
  - La tâche libère le verrou en fin de traitement; il redevient ouvert.
  - Si le verrou est fermé lors de l'acquisition par une tâche, cette dernière est suspendue jusqu'à ce que le verrou redevienne ouvert.
- Le mutex est un mécanisme d'attente passive.
  - Basé sur le principe qu'il est inutile d'activer une tâche en attente de la section critique.
  - La libération de la section critique est mise à profit pour relancer une des tâches en attente.

Mutex

- Le Mutex
  - est une variable booléenne
  - possède une liste d'attente
  - est manipulé par deux opérations *atomiques*:
    - `Verrouille(v)`

```
void Verrouille(verrou v)
{
  if (v)
    v = false;
  else
    suspendre la tâche appelante dans la file associée à v
}
```

    - `Deverrouille(v)`

```
void Déverrouille(verrou v)
{
  if (la file associée à v != vide)
    débloquer une tâche en attente dans file
  else
    v = true;
}
```


API Xenomai - Mutex

Fonction	Description	Kern.	User
`int rt_mutex_create(RT_MUTEX *mutex,const char *name)`	Création d'un verrou (mutex)	✓	✓
`int rt_mutex_acquire(RT_MUTEX *mutex, RTIME timeout)`	Acquisition d'un verrou	✓	✓
`int rt_mutex_release(RT_MUTEX *mutex)`	Restitution d'un verrou	✓	✓
`int rt_mutex_delete(RT_MUTEX *mutex)`	Destruction d'un verrou	✓	✓

- L'appel à **`rt_mutex_acquire`**`()` est potentiellement bloquant
- Un mutex Xenomai est récursif
- La file d'attente est gérée par priorité + FIFO

Accès concurrent - Exemple

Global

```
static int count=0;
RT_MUTEX mutex;
```

```
void taskT1(void *cookie) {
  while (1) {
    rt_mutex_acquire(&mutex,
                     TM_INFINITE);
    /* Accès à une ressource */
    /* commune */
    count = count + 1;

    /* Accès à
un périphérique */
    iowrite(GPIO_1, "blabla");
    rt_mutex_release(&mutex);
    rt_task_wait_period(NULL);
  }
}
```

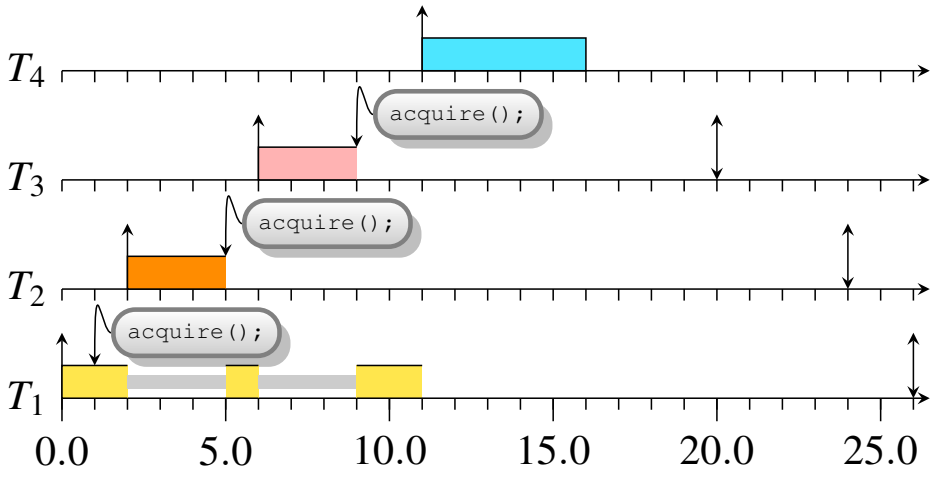
```
void taskT2(void *cookie) {
  int value;
  while (1) {
    rt_mutex_acquire(&mutex,
                     TM_INFINITE);
    /* Accès à une ressource */
    /* commune */
    count = count + 1;

    /* Accès à
un périphérique */
    value=ioread(GPIO_1);
    rt_mutex_release(&mutex);
    rt_task_wait_period(NULL);
  }
}
```


Mutex - Exemple

- Politique de la file d'attente: FIFO
- Un seul mutex
- Chaque tâche libère le mutex en fin d'exécution
- $Pr(T_4) > Pr(T_3) > Pr(T_2) > Pr(T_1)$
- Complétez le chronogramme suivant:

T	C
T_4	5
T_3	5
T_2	5
T_1	7

Que faudrait-il faire pour que le système soit ordonnançable sur les 26 premières unités de temps?

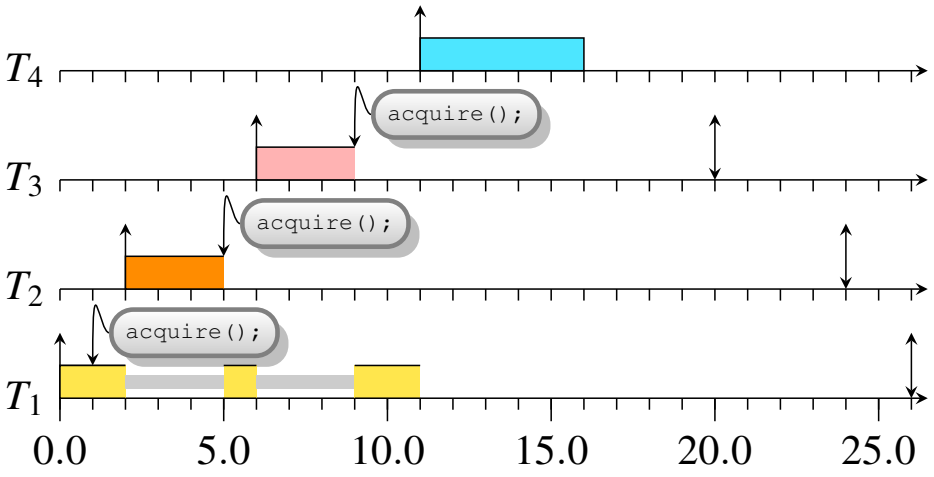


Mutex - Exemple (2)

- Politique de la file d'attente: par priorité
- Un seul mutex
- Chaque tâche libère le mutex en fin d'exécution
- $Pr(T_4) > Pr(T_3) > Pr(T_2) > Pr(T_1)$
- Complétez le chronogramme suivant:

T	C
T_4	5
T_3	5
T_2	5
T_1	7

Que faudrait-il faire pour que le système soit ordonnançable sur les 26 premières unités de temps?



Quand utiliser un mutex

- Lorsque plusieurs tâches accèdent à une même variable partagée pas uniquement en lecture
- Pour protéger l'accès à une ressource matérielle
  - Port de communication
  - ADC
  - ...
- Certaines fonctions offertes par les librairies sont *réentrantes*
  - Signifie qu'elles peuvent être appelées par plusieurs tâches simultanément
  - Certaines fonctions offrent une version réentrante et une non réentrante
  - Une directive de pré-compilation -DREENTRANT permet de forcer l'utilisation de la réentrante

Sémaphores

- Les sémaphores sont une généralisation des verrous
- Proposés par Dijkstra en 1965
- comprennent une variable entière plutôt qu'un booléen
- Opérations d'accès (atomiques):
  - $P(s)$ (pour tester: Prolaag (probeer te verlagen: essayer de réduire))
  - $V(s)$ (pour incrémenter: Verhoog)

Sémaphores

- Pseudocode des deux opérations

```
void P(sémaphore s)
{
  s -= 1;
  if (s < 0)
    suspendre la tâche appelante dans la file associée à s
}

void V(sémaphore s)
{
  s += 1;
  if (s <= 0)
    débloquer une des tâches de la file associée à s
}
```

Sémaphore - Introduction

- Un sémaphore permet à plusieurs tâches de rentrer dans une section critique commune.
  - Un périphérique pourrait accepter un nombre limité de requêtes simultanées.
- A la différence d'un mutex, un sémaphore n'est pas détenu par une tâche (en effet, plusieurs tâches peuvent entrer dans la section critique, en utilisant le même sémaphore).
- Trois opérations de base
  - **`rt_sem_create`**`(&sema, .., size, ...)` création d'un sémaphore d'identificateur n avec size entrées
  - **`rt_sem_p`**`(&sema, ...)` attendre sur le sémaphore n (Puis-je?)
  - **`rt_sem_v`**`(&sema)` signaler le sémaphore n (Vas-y!)

API Xenomai - Sémaphores

Fonction	Description	Kern.	User
`int rt_sem_create(RT_SEM *sem, const char *name, unsigned long icount, int mode)`	Création d'un sémaphore	✓	✓
`int rt_sem_p(RT_SEM *sem, RTIME timeout)`	Réservation d'une entrée de sémaphore	✓	✓
`int rt_sem_v(RT_SEM *sem)`	Libération d'une entrée de sémaphore	✓	✓
`int rt_sem_delete(RT_SEM *sem)`	Destruction d'un sémaphore	✓	✓

- L'appel à **`rt_sem_p`**`()` est potentiellement bloquant

Section partagée

- Les sémaphores peuvent être exploités pour protéger une section partagée
- Exemple: 2 tâches ont droit d'accéder à une section partagée

```
RT_SEM sem;

void *T1(void *arg) {
  rt_sem_p(&sem,TM_INFINITE);
  // section partagée
  rt_sem_v(&sem);
}

void main() {
  // init. du sémaphore à 2
  rt_sem_create(&sem,"Semaphore",
                2,S_PRIO);
  // création des taches
}
```

```
void *T2(void *arg) {
  rt_sem_p(&sem,TM_INFINITE);
  // section partagée
  rt_sem_v(&sem);
}

void *T3(void *arg) {
  rt_sem_p(&sem,TM_INFINITE);
  // section partagée
  rt_sem_v(&sem);
}
```


Synchronisation à base de sémaphores (1)

- Les sémaphores peuvent être exploités pour synchroniser des tâches
- Exemple: T_1 doit attendre que T_2 et T_3 aient terminé

```
RT_SEM sem;

void *T1(void *arg) {
  rt_sem_p(&sem,TM_INFINITE);
  rt_sem_p(&sem,TM_INFINITE);
  // T2 et T3 ont terminé
}

void main() {
  rt_sem_create(&sem,"Semaphore",
                0,S_PRIO);
  // création des taches
}
```

```
void *T2(void *arg) {
  // traitement

  rt_sem_v(&sem);
}

void *T3(void *arg) {
  // traitement

  rt_sem_v(&sem);
}
```

Synchronisation à base de sémaphores (2)

- Exemple: T_1 permet à T_2 et T_3 de continuer

```
RT_SEM sem;

void *T1(void *arg) {
  // traitement

  rt_sem_v(&sem);
  rt_sem_v(&sem);
}

void main() {
  rt_sem_create(&sem,"Semaphore",
                0,S_PRIO);
  // création des taches
}
```

```
void *T2(void *arg) {
  rt_sem_p(&sem,TM_INFINITE);
  // traitement

}

void *T3(void *arg) {
  rt_sem_p(&sem,TM_INFINITE);
  // traitement

}
```


Gestion des conditions (prédicats)

- Une tâche peut éviter de gaspiller du temps CPU inutilement en attendant qu'une condition devienne vraie.

Tâche A

```
void taskA(void *cookie) {
  while (1) {
    rt_mutex_acquire(&verrou,
                     TM_INFINITE);
    if (count == 100) {
      /* Do something ... */
      count--;
    }
    rt_mutex_release(&verrou);
    rt_task_wait_period(NULL);
  }
}
```

Tâche B

```
void taskB(void *cookie) {
  while (1) {
    rt_mutex_acquire(&verrou,
                     TM_INFINITE);
    count++;



    rt_mutex_release(&verrou);
    rt_task_wait_period(NULL);
  }
}
```

- $Pr(T_a) > Pr(T_b)$

Variable condition (1/3) - Mécanisme

- On appelle *variable condition* une variable qui peut suspendre une tâche jusqu'à ce qu'une condition (prédicat) devienne vraie.
- La suspension de la tâche s'effectue à l'aide de l'opération **`rt_cond_wait`**`(var)`, alors qu'une autre tâche utilisera l'opération **`rt_cond_signal`**`(var)` pour réveiller la tâche en attente.
- Remarque: ce mécanisme permet l'implantation de moniteurs.

Variable condition (2/3) - Principes

- Une variable condition n'est modifiable qu'avec l'utilisation des primitives **`rt_cond_wait`**`()` et **`rt_cond_signal`**`()`.
- **`rt_cond_wait`**`(x)` bloque l'exécution de la tâche sur la condition x.
  - la tâche reprendra l'exécution seulement si une autre exécute **`rt_cond_signal`**`(x)`.
  - **`rt_cond_wait`**`(x)` est toujours bloquant.

Variable condition (3/3) - Principes

- **`rt_cond_signal`**`(x)` reprend l'exécution d'une tâche bloquée sur la condition x.
  - S'il existe une tâche en attente, elle sera débloquée (passage à READY).
    - S'il existe plusieurs tâches en attente, c'est la politique de la file d'attente associée à la variable condition qui "dira" quelle tâche doit être exécutée (par priorité + FIFO pour Xenomai); les autres tâches restent dans l'état WAITING.
    - L'opération n'est valable que pour une seule tâche (mode de réveil normal).
    - Il existe une autre fonction (**`rt_cond_broadcast`**`(x)`)pour le mode broadcast.
  - S'il n'en existe pas: ne rien faire, l'événement est considéré comme perdu (pas de mémorisation du signal de l'événement).

API Xenomai - Variables condition

Fonction	Description	Kern.	User
`int rt_cond_create(RT_COND *cond, const char *name)`	Création d'une variable condition	✓	✓
`int rt_cond_wait(RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)`	Suspend la tâche jusqu'à ce que la variable condition soit signalée	✓	✓
`int rt_cond_signal(RT_COND *cond)`	Signalement d'une variable condition	✓	✓
`int rt_cond_broadcast(RT_COND *cond)`	Signalement d'une variable condition pour toutes les tâches en attente	✓	✓
`int rt_cond_delete(RT_COND *cond)`	Destruction d'un sémaphore	✓	✓

- L'appel à **`rt_cond_wait`**`()` est forcément bloquant

Variable condition (1/2) - Mutex

- Il peut y avoir un effet indésirable avec l'utilisation d'une variable condition.

Tâche A

```
...
if (!empty)
  ←~ Changement de contexte
  rt_cond_wait(&x, &mutex,
               TM_INFINITE);

/* C'est vide ! */
```

Tâche B

```
...

/* C'est vide */
empty=1;
rt_cond_signal(&x);
```

$$Pr(TacheB) > Pr(TacheA)$$

- Un mutex est associé à la variable condition afin de garantir un état consistent.
- La variable ne peut pas changer d'état durant la préparation au wait().
- La tâche ne peut reprendre l'exécution que si elle détient le mutex associé.

Variable condition (2/2) - Mutex

- On peut supprimer l'effet indésirable en effectuant le changement d'état ET l'envoi du signal à l'intérieur d'une section critique, encadrée par un mutex.

Tâche A

```
...
rt_mutex_acquire(&mutex, TM_INFINITE);

if (!check_if_empty)

  rt_cond_wait(&x, &mutex, TM_INFINITE);

rt_mutex_release(&mutex);

/* C'est vide ! */
```

Tâche B

```
...
rt_mutex_acquire(&mutex,
                 TM_INFINITE);

check_if_empty = 1;

rt_cond_signal(&x);

rt_mutex_release(&mutex);
...
```

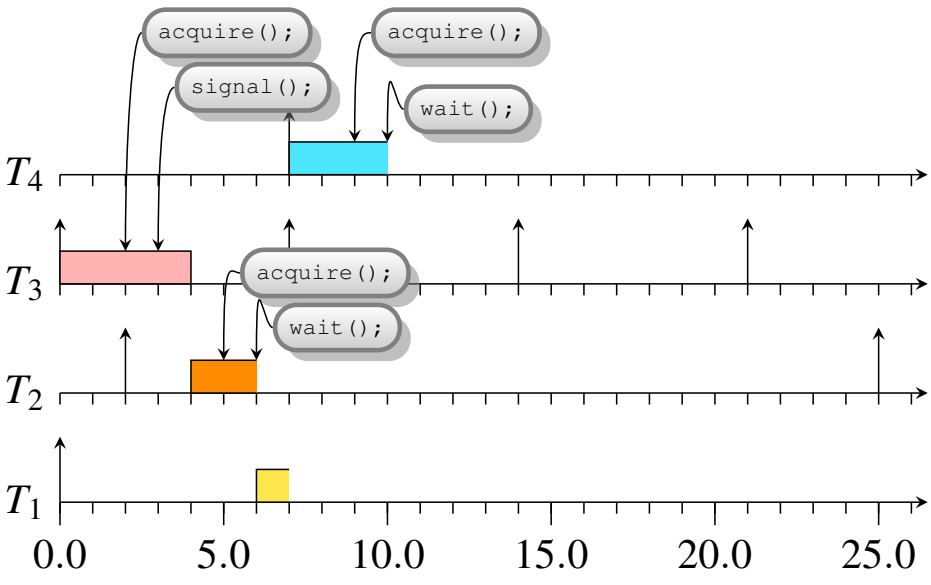
Variable condition - Mutex

- Lorsqu'une tâche se met en attente sur une variable condition:
  - Elle relâche le mutex associé
- Lorsque la tâche est réactivée suite à un signal sur la condition
  - Elle doit réacquérir le mutex
    - Elle est alors en compétition avec les autres tâches

Variable condition - Exemple (1)

- Politique de la file d'attente: par priorité
- Une seule variable condition, associée à un mutex
- Chaque tâche libère le mutex en fin d'exécution
- $Pr(T_4) > Pr(T_3) > Pr(T_2) > Pr(T_1)$. T_3 est périodique.
- Complétez le chronogramme suivant:

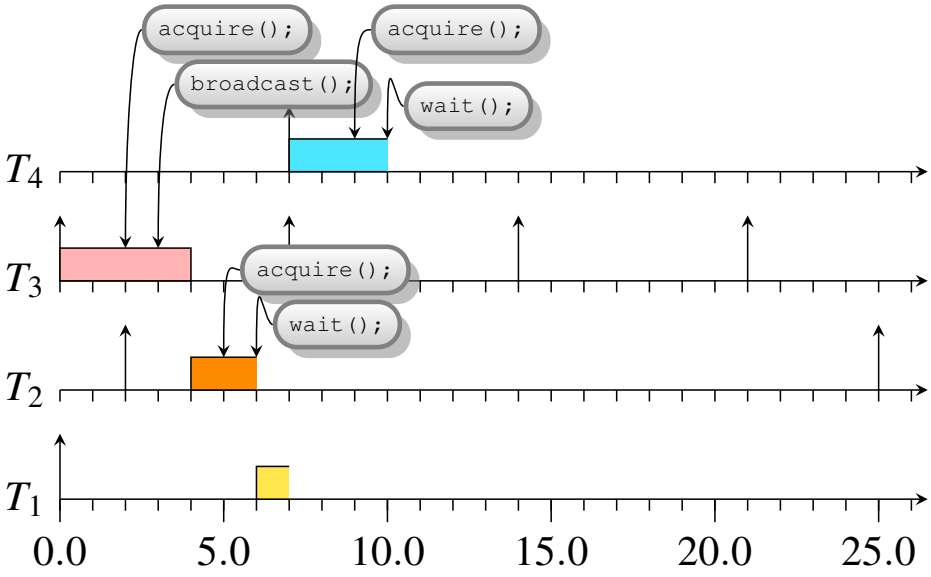
T	C
T_4	5
T_3	4
T_2	5
T_1	3



Variable condition - Exemple (1)

- Politique de la file d'attente: par priorité
- Une seule variable condition, associée à un mutex
- Chaque tâche libère le mutex en fin d'exécution
- $Pr(T_4) > Pr(T_2) > Pr(T_3) > Pr(T_1)$. T_3 est périodique.
- Complétez le chronogramme suivant:

T	C
T_4	5
T_3	4
T_2	5
T_1	3



Inversion de priorité - Description

- L'inversion de priorité est un problème lié à un accès concurrent à une ressource.
  - Une tâche prioritaire est bloquée par une tâche moins prioritaire, car cette dernière détient une ressource désirée par la tâche prioritaire.
  - De plus une troisième tâche vient s'intercaler entre deux.
    - Cette tâche n'accède pas la ressource partagée entre les deux premières, mais néanmoins bloque les deux tâches!

Inversion de priorité (1/2) - Exemple

Tâche	Priorité	Capacité	Arrivée	mutex_acq.	mutex_rel.
taskA	1	11	2	2	9
taskB	2	6	12	-	-
taskC	3	7	7	3	4

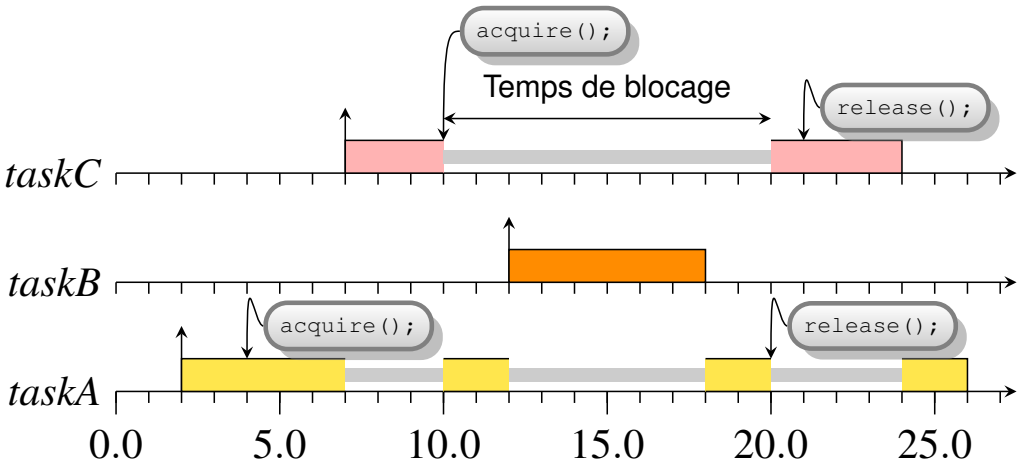
```
void taskA(void *cookie) {

  while (1) {

    /* Doit accéder GPIO_1 */
    rt_mutex_acquire(&verrou,
                     TM_INFINITE);
    /* ... */

    rt_mutex_release(&verrou);

    rt_task_wait_period(NULL);
  }
}
```

```
void taskB(void *cookie) {


  while (1) {


    /* Gère les communications
       */
    ...


    rt_task_wait_period(NULL);
  }
}
```

```
void taskC(void *cookie) {

  while (1) {
    /* Préparation */
    ...
    /* Doit accéder GPIO_1 */
    rt_mutex_acquire(&verrou,
                     TM_INFINITE);
    /* ... */
    rt_mutex_release(&verrou);

    rt_task_wait_period(NULL);
  }
}
```


Inversion de priorité (2/2) - Exemple

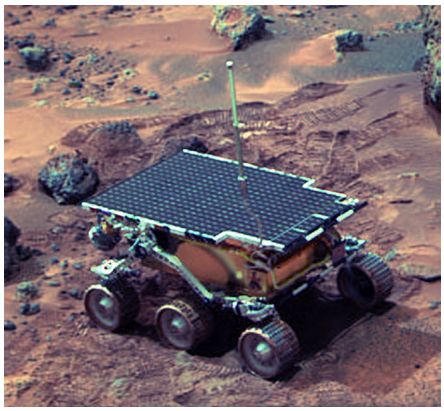
Tâche	Priorité	Capacité	Arrivée	mutex_acq.	mutex_rel.
taskA	1	11	2	2	9
taskB	2	6	12	-	-
taskC	3	7	7	3	4



- Quel sera le temps de réponse de la tâche "taskC"?
- Que se passerait-il s'il y avait 3 ou 4 tâches "taskB"?

Inversion de priorité - Pathfinder

- Est-ce bien utile de se préoccuper de l'inversion de priorité lors du développement d'applications temps-réel?
  - Exemple: Mission Pathfinder



  - Une inversion de priorité est à l'origine de redémarrages intempestifs du système!!!
    - `http://perso.wanadoo.fr/degeeter/inv_fr.htm`

Inversion de priorité - Problème

- Une gestion de files d'attente par priorité ne résout pas le problème de l'inversion de priorité.
- Il peut y avoir beaucoup de tâches de priorité intermédiaire.
  - Il est difficile d'estimer la durée de blocage d'une tâche à haute priorité.
  - $\Rightarrow$ Les contraintes temps-réel ne peuvent plus être garanties!

Inversion de priorité - Solution

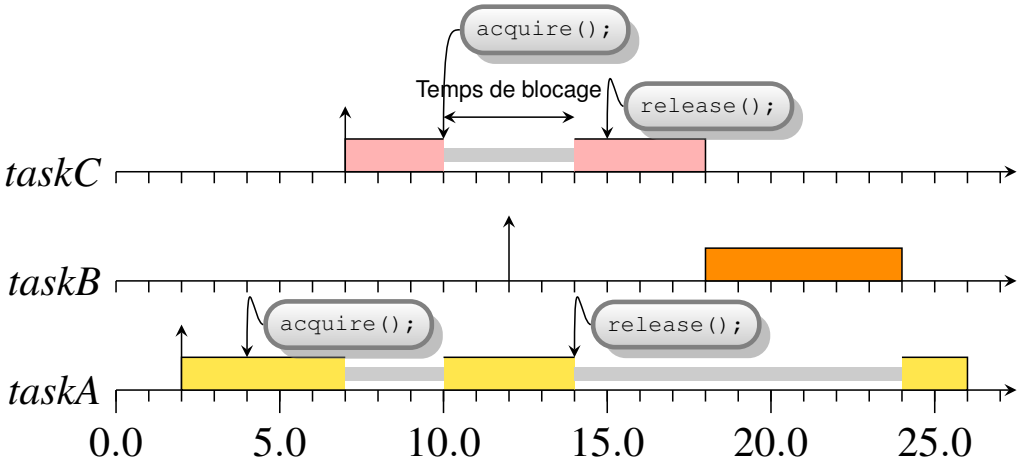
- Protocole d'héritage simple de priorité
  - PIP: Priority Inheritance Protocol
- Protocole à priorité plafond
  - PCP: Priority Ceiling Protocol

PIP - Description

- Prérequis
  - Ordonnancement à priorité fixe (priorité nominale)
  - Mutex pour sections critiques
- Définition du protocole
  - Règle d'ordonnancement
    - par défaut, ordonnancement selon priorité nominale
  - Règle d'allocation de ressource
    - Ressource libre: acquisition de la ressource
    - Ressource occupée: blocage
  - Règle d'héritage de priorité
    - Quand il y a blocage sur une ressource, la priorité de la tâche qui la possède est augmentée au niveau de la priorité de la tâche qui la réclame (priorité supérieure par définition). Lorsque la ressource est libérée, la tâche reprend la priorité qu'elle avait avant l'acquisition de la ressource.

PIP - Exemple

Tâche	Priorité	Capacité	Arrivée	mutex_acq.	mutex_rel.
taskA	1	11	2	2	9
taskB	2	6	12	-	-
taskC	3	7	7	3	4



- Le temps de blocage est borné par la durée de la section critique

PIP - Propriétés

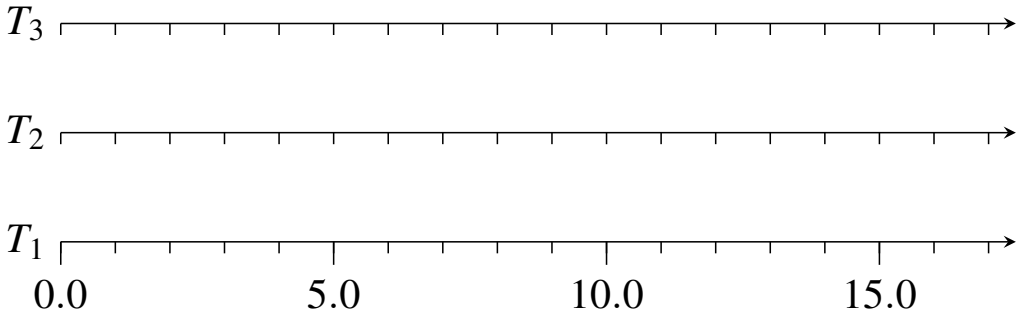
- Temps de blocage borné
- Interblocages possibles
- Pour implanter PIP, il n'est pas nécessaire de savoir quelle tâche utilise quelle ressource.
  - Par contre, nécessaire pour calculer les temps de blocage.
- N'élimine PAS l'inversion de priorité, mais la rend juste de durée bornée.
- L'utilisation d'une politique de file d'attente par priorité est nécessaire.
  - Respect des priorités entre tâches.

Inversion de priorité - Blocage

- Temps de blocage
  - Durée pendant laquelle une tâche T ne peut plus progresser à cause d'une tâche moins prioritaire.
- Sources de blocage
  - Blocage direct
    - T partage une ressource avec une tâche moins prioritaire qu'elle.
    - Relation transitive en cas de sections critiques imbriquées.
    - Si T partage une ressource R avec T' moins prioritaire, si la section critique sur R inclut l'utilisation d'une ressource R' et si R' est utilisée par une tâche T" moins prioritaire que T', alors T' dépend de T" et par transitivité, T dépend de T".
  - Blocage indirect (pass-through blocking)
    - T est bloquée par une tâche moins prioritaire, qui ne partage pas de ressource avec T, mais qui a hérité d'une priorité plus importante que T via le mécanisme d'héritage de priorité.
    - Une tâche n'utilisant pas de ressource partagée peut subir un temps de blocage non nul par ce phénomène.

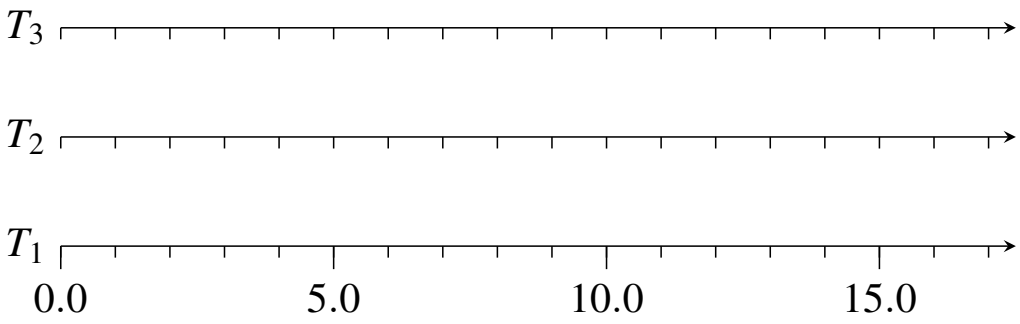
Exemple de blocage direct

T	Arr.	C	Pr	req R1	rel R1	req R2	rel R2
T_3	6	3	3	2	3	-	-
T_2	3	5	2	1	4	2	3
T_1	0	8	1	-	-	2	6



Exemple de blocage indirect

T	Arr.	C	Pr	req R	rel R
T_3	3	3	3	1	2
T_2	2	3	2	-	-
T_1	0	6	1	1	4



Temps de blocage (1/2)

- La durée maximale de blocage Bi de la tâche i est égale à la somme des durées des sections critiques partagée par la tâche i et les tâches de priorité inférieure.
  - Une première approximation de la durée maximale peut être donnée par:

$$B_i = \sum_k usage(k,i) CS_k$$

Avec, pour une section critique k:

$$usage(k,i) = \begin{cases} 1 & \text{s'il existe au moins 1 tâche} \\ & \text{de priorité < Pr(Ti) ET} \\ & \text{au moins 1 tache de priorité >=Pr(Ti)} \\ 0 & \text{sinon} \end{cases}$$

Temps de blocage (2/2)

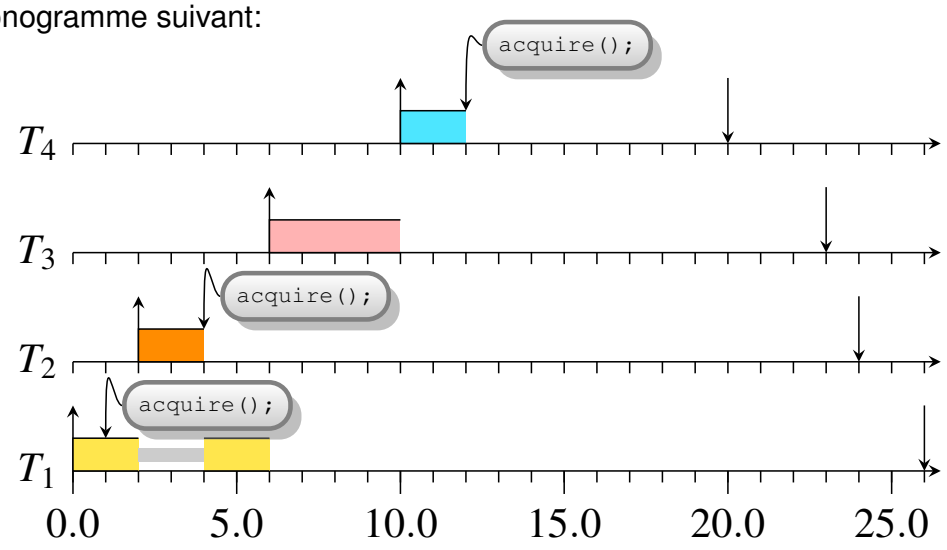
- Avec ce qui précède, le calcul du temps de réponse doit tenir compte des blocages des sections critiques.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Inversion de priorité & PIP

- Politique de la file d'attente: par priorité
- Un seul mutex implémentant le protocole d'héritage de priorité (PIP)
- Chaque tâche libère le mutex en fin d'exécution, à l'exception de T_1 qui le libère une unité avant de terminer
- $Pr(T_4) > Pr(T_3) > Pr(T_2) > Pr(T_1)$
- Complétez le chronogramme suivant:

T	C
T_4	5
T_3	5
T_2	5
T_1	10



Héritage de priorité - Remarques

- Généralement, les exécutifs temps-réel offrent un service de mutex muni d'un mécanisme d'héritage de priorité.
  - Transparent pour le développeur
  - Le protocole implémenté est souvent le PIP
- Attention, les sémaphores n'implémentent pas forcément ce protocole

Priority Ceiling Protocol - Description

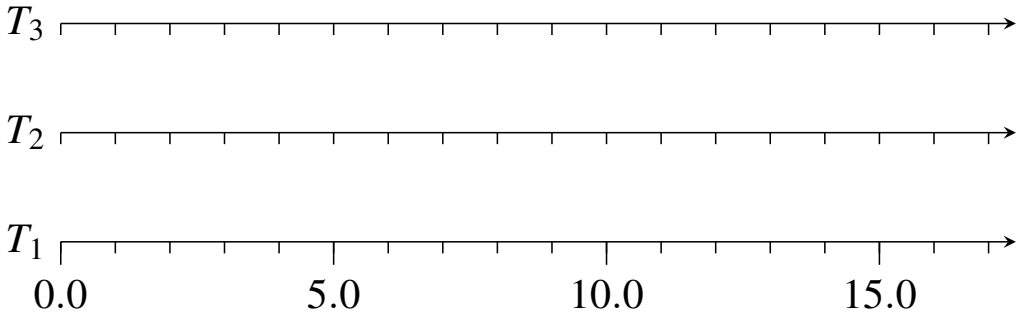
- Prérequis
  - Ordonnancement à priorité fixe (priorité nominale)
  - Mutex pour sections critiques
- Définition du protocole
  - Règle d'ordonnancement
    - par défaut, ordonnancement selon priorité nominale
  - Règle d'allocation de ressource
    - Soit $PC(S) =$ priorité de la tâche la plus prioritaire pouvant faire appel à la ressource S
    - Ressource libre: une tâche T de priorité P peut accéder à la ressource si P est strictement plus grand que tous les PC de toutes les ressources actuellement réquisitionnées par d'autres tâches
    - Ressource occupée: blocage
  - Règle d'héritage de priorité
    - Idem à PIP (modification)

PCP

- Les priorités de plafond sont définies de manière statique
- Avantage: élimine les deadlocks

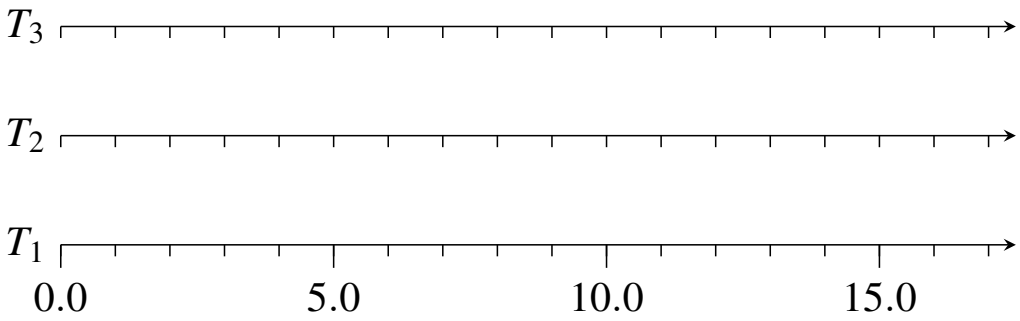
Exemple de blocage direct

T	Arr.	C	Pr	req R1	rel R1	req R2	rel R2
T_3	6	3	3	2	3	-	-
T_2	3	5	2	1	4	2	3
T_1	0	8	1	-	-	2	6



Exemple de blocage direct - PCP

T	Arr.	C	Pr	req R1	rel R1	req R2	rel R2
T_3	6	3	3	2	3	-	-
T_2	3	5	2	1	4	2	3
T_1	0	8	1	-	-	2	6

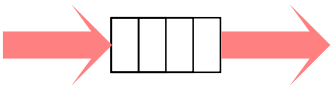


Les temps de relâchement de mutex sont relatifs au temps de début d'exécution de la tâche

Communication - Introduction (1/2)

- Les tâches temps-réel peuvent s'échanger des données afin d'accomplir leurs objectifs.
  - Ces données peuvent prendre la forme d'un message; celui-ci est créé par le noyau, délivré à une ou plusieurs tâches et effacée. L'ordre d'arrivée et de délivrance des messages est important.
  - Le noyau offre un service de communication (ou de gestion de messages).
- Les principaux mécanismes de communication reposent sur la notion de *file (ou queue) de messages*.
  - On parle aussi de boîte au lettres: un message est déposé dans un endroit réservé (zone mémoire) à cet effet. La tâche réceptrice consultera alors la boîte aux lettres pour voir si un message est disponible.

Communication - Introduction (2/2)

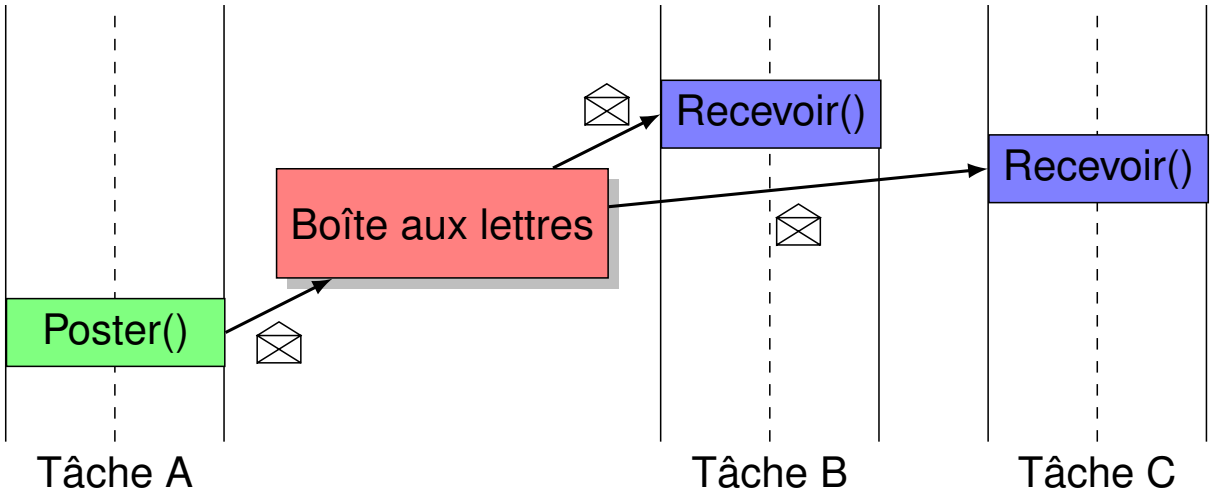
- Modèles fonctionnels d'une communication par message:
  - Communication synchrone: T_A  T_B
  - Communication asynchrone: T_A  T_B
- Si la communication se fait directement de tâche à tâche, on parle de communication synchrone; les tâches s'attendent mutuellement.
  - On parle aussi de rendez-vous.
- Dans la majorité des cas, la communication est asynchrone: la communication nécessite la présence d'une mémoire tampon.
  - Cette mémoire tampon portera le nom de boîte au lettres, ou file de messages, ou queue de messages.
  - Cette file est généralement bornée: un nombre maximum de messages peut y prendre place.

Communication - Caractéristiques

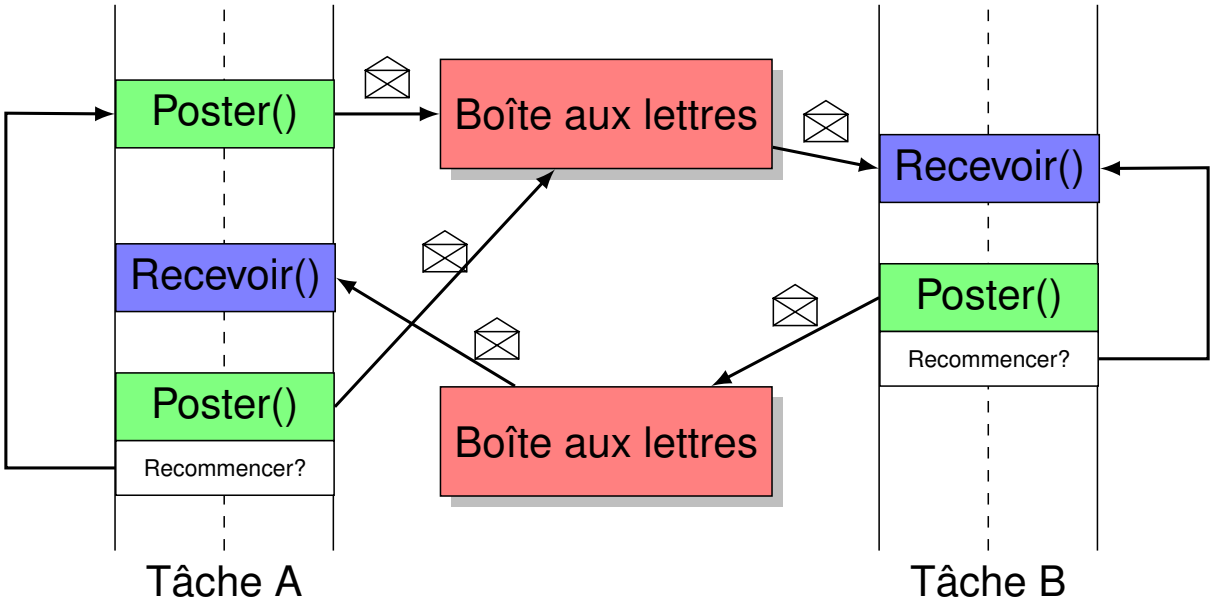
- Le modèle de communication asynchrone nécessite les éléments suivants:
  - Un message
    - Caractéristiques: taille, structure prédéfinie (date, taille effective, etc.)
  - Une file de messages
    - Caractéristiques: nom, taille (totale), nombre de messages maximum
  - Une file d'attente
    - Les tâches en attente d'un message sont placées dans une file d'attente (état WAITING). L'ordre dans lequel les tâches recevront le message dépendra de la politique de la file d'attente.
    - Politique FIFO: la première tâche en attente sera la première servie.
    - Politique "par priorité": la tâche la plus prioritaire sera desservie.
- L'envoi d'un message s'accompagne d'un mode de réveil des tâches en attente.
  - Mode normal: une seule tâche en attente est informée de la présence d'un message.
  - Mode broadcast: toutes les tâches en attente d'un message sont informées de la présence d'un message.

File de message - Exemple (1/2)

- Un message peut être diffusé à toutes les tâches en attente (le mode de réveil est alors le mode broadcast).



File de message - Exemple (2/2)



API Xenomai - File de messages

Fonction	Description	Kern.	User
`int rt_queue_create(RT_QUEUE *q, const char *name, size_t poolsize, size_t qlimit, int mode)`	Création d'une file de messages	✓	✓
`void *rt_queue_alloc(RT_QUEUE *q, size_t size)`	Allocation d'un message (géré par un pool de messages)	✓	✓
`int rt_queue_send(RT_QUEUE *q, void *buf, size_t size, int mode)`	Envoi d'un message	✓	✓
`size_t rt_queue_receive(RT_QUEUE * q, void ** bufp, RTIME timeout)`	Réception d'un message	✓	✓
`int rt_queue_free(RT_QUEUE *q, void *buf)`	Restitution de la place mémoire utilisée pour le message (côté récepteur)	✓	✓
`int rt_queue_delete(RT_QUEUE *q)`	Destruction d'une file de messages	✓	✓
`int rt_queue_write(RT_QUEUE *q,const void *buf, size_t size,int mode)`	Ecriture d'un message	✓	✓
`size_t rt_queue_read(RT_QUEUE *q,void *buf, size_t size,RTIME timeout)`	Lecture d'un message	✓	✓

- L'appel à **`rt_queue_receive`**`()` et **`rt_queue_write`**`()` est potentiellement bloquant

Exemple d'utilisation (1/2)

```
RT_QUEUE mailbox;

void taskProducer(void *cookie) {
  char *msg;
  msg = rt_queue_alloc(&mailbox, 20);
  strcpy(msg, "Hello world!");
  rt_queue_send(&mailbox, msg, strlen(msg)+1, Q_NORMAL);
  ...
}

void taskConsumer(void *cookie) {
  char *msg;
  rt_queue_receive(&mailbox, (void **) &msg, TM_INFINITE);
  rt_queue_free(&mailbox, msg);
  ...
}

int main (void) {
  int err;
  err = rt_queue_create(&mailbox, "myMailbox", 80, 1, Q_FIFO);
  if (err != 0)
    perror("mailbox creation failed\n");
  ...
  rt_task_spawn(&producerTask, "ProducerTask", STACK_SIZE, 1, 0, producer, NULL);
  rt_task_spawn(&consumerTask, "ConsumerTask", STACK_SIZE, 1, 0, producer, NULL);
  ...
}
```

Une tâche réveillée

taille du pool

1 message

Exemple d'utilisation (2/2)

```
RT_QUEUE mailbox;

void taskProducer(void *cookie) {
  char *msg;
  msg = rt_queue_alloc(&mailbox, 20);
  strcpy(msg, "Hello world!");
  rt_queue_send(&mailbox, msg, strlen(msg)+1, Q_BROADCAST);
  ...
}

void taskConsumer(void *cookie) {
  char *msg;
  rt_queue_receive(&mailbox, (void **) &msg, TM_INFINITE);
  rt_queue_free(&mailbox, msg);
  ...
}

int main (void) {
  int err;
  err = rt_queue_create(&mailbox, "myMailbox", 1024, 5, Q_FIFO);
  if (err != 0)
    perror("mailbox creation failed\n");
  ...
  rt_task_spawn(&producerTask, "ProducerTask", STACK_SIZE, 1, 0, producer, NULL);
  rt_task_spawn(&consumerTask, "ConsumerTask", STACK_SIZE, 1, 0, producer, NULL);
  ...
}
```

Toutes les tâches réveillées

taille du pool

5 messages

Communication inter-processus

- Si différents processus doivent échanger de l'information, deux options offertes par Xenomai:
  - Buffer
  - Message Pipe

Buffer

- Un buffer mémoire est créé par un processus, puis *bindé* par l'autre processus
- Ecriture: un pointeur et une taille
- Lecture: un pointeur et une taille

Message Pipe

- Permet une communication bi-directionnelle entre deux processus
- Fonctionne pour un processus Xenomai et un processus Linux standard
- Côté Linux, ouverture du fichier `/dev/rtpN` où `N` est le numéro mineur donné à la création du pipe
- Côté Xenomai, création du pipe, puis accès via des messages ou un couple (pointeur,taille)

Allocation mémoire - Principes

- Dans un système temps-réel, l'allocation mémoire est généralement statique (une fois pour toutes, au début du programme).
  - Pas de "mauvaise surprise" sur la disponibilité mémoire (pas de Memory Overflow).
  - Pas d'algorithme d'allocation mémoire compliqué nécessitant beaucoup de temps de calcul.
    - On veut limiter le décalage temporel dû à l'allocation mémoire autant que possible.
- Toutefois, il est possible de disposer de mécanismes d'allocation dynamique pour optimiser l'utilisation de la mémoire.

Notion de *pool* mémoire (ou *tas* mémoire)

- Les noyaux temps-réel offrent la possibilité d'allouer un pool mémoire dans lequel l'allocateur mémoire "puisera" des blocs mémoire à la demande.
- Ce pool mémoire est appelé tas mémoire (memory heap).
  - Un bloc mémoire peut être alloué à partir d'un tas prédéfini.
  - Le bloc est alloué complètement à une tâche, ou ne l'est pas.
  - Un tas peut être créé dynamiquement à n'importe quel moment, généralement au début du programme.
  - La taille du tas est fixe, les blocs demandés peuvent être de taille différente.
  - Les blocs mémoire sont alloués/restitués dynamiquement, au fil des besoins.
- Cette méthode par allocation à l'aide d'un pool permet de garantir un déterminisme temporel.

API Xenomai - Allocation mémoire

Fonction	Description	Kern.	User
`int rt_heap_create(RT_HEAP *heap, const char *name, size_t heapsize, int mode)`	Création d'un pool de bloc dans le tas (mode=H_PRIO ou H_FIFO)	✓	✓
`int rt_heap_alloc(RT_HEAP *heap, size_t size, RTIME timeout, void **blockp)`	Allocation d'un bloc mémoire à partir du pool précédemment créé	✓	✓
`int rt_heap_free(RT_HEAP *heap, void *block)`	Restitution d'un bloc mémoire	✓	✓
`int rt_heap_delete(RT_HEAP *heap)`	Destruction d'un pool mémoire	✓	✓

- L'appel à **`rt_heap_alloc()`** est potentiellement bloquant

Allocation mémoire

Exemple

```
RT_HEAP myHeap;

void myTask(void *) {
  // Do something
  char *buffer;
  if (rt_heap_alloc(&myHeap, 10, TM_INFINITE, &buffer) != 0)
    ERROR();
  // Now we can use buffer

  // And free it
  rt_heap_free(&myHeap, buffer);
}

int main (int argc, char *argv[]) {
  if (rt_heap_create(&myHeap, "My heap", 1000, H_PRIO) != 0) {
    ERROR();

  // Create the task

  // Join

  rt_heap_delete(&myHeap);
}
```